Here we extend our language with a simple I/O mechanism that allows us to initialize variables and examine variables in the executing program:

\[
C ::= \text{put}(A) \mid \text{get}(x)
\]

The informal semantics is that 'put' allows us to write an expression to the terminal and 'get' allows us to initialize a declared variable with an integer value read from the terminal.
The formal semantics is as follows:

\[\text{(put}(A),\text{State}) \rightarrow\rightarrow \text{State} : - \quad \text{\%io\% writing}\]
\[\text{(A,State) } \rightarrow\rightarrow \text{ ValA,}\]
\[\text{write}(A),\]
\[\text{write(’ is ’)},\]
\[\text{writeln}(\text{ValA}),!\].

\[\text{(get}(X),\text{State}) \rightarrow\rightarrow \text{ OState} : - \quad \text{\%io\% reading}\]
\[\text{lookup}(X,\text{State},\_),\]
\[\text{write(’Enter integer value for ’)},\]
\[\text{write}(X),\]
\[\text{write(’: ’)},\]
\[\text{read}(\text{Val}),\]
\[\text{int}(\text{Val}),\]
\[\text{bindval}(X,\text{Val},\text{State},\text{OState}),!\].
Now we can write programs such as these:

?- ['sem-block.pl'].
% xis.pl compiled 0.01 sec, 7,792 bytes
% preamble.pl compiled 0.01 sec, 8,956 bytes
% xis.pl compiled 0.00 sec, 148 bytes
% sem-block.pl compiled 0.01 sec, 18,284 bytes
true.

?- ((var(x) seq get(x) seq put(x)),s) -->> V.
Enter integer value for x: 3.
x is 3
V = push([bind(3, x)], s).

?- ((var(x) seq get(x) seq put(add(x,1))),s) -->> V.
Enter integer value for x: 5.
add(x,1) is 6
V = push([bind(5, x)], s).

?-
In most languages a block introduces a new scope allowing for *local variable declarations*. In C blocks are introduced with the curly braces,

```c
{ 
    int x;
}
```

We can access the values of variables in non-local scope. Consider the following code,

```c
{ 
    int x = 2;
    {
        int y = 3;
        x = y + x; /* accessing the surrounding scope via 'x' */
    }
}
```

In most languages blocks can be nested ⇒ *variable shadowing*.

```c
{ 
    int x = 1;
    {
        int x = 2; /* the original 'x' is no longer visible in this scope */
    }
}
```
Recall that in our simple language we have variable declarations and now we introduce blocks:

\[ C ::= \text{var}(x) \mid \text{block}(C) \]

Think of 'block' as 'begin C end' where C could be any command including sequential composition.
Going back to our observations on block structured languages

- **Local variable declarations.** When we leave a scope with local variables those variables should become undeclared:
  
  ```
  var(x) seq block( var(y) seq assign(y,1) ) seq assign(y,x)
  ```

- **Non-local side effects.** When assigning a value to a variable declared in a surrounding scope we need to update the value of that variable, the value printed out for x should be 2:
  
  ```
  var(x) seq assign(x,1) seq block( assign(x,2) ) seq put(x)
  ```

- **Variable shadowing.** Redeclaring a variable in a nested scope with the same name as a variable in the outer scope makes the variable in the outer scope unavailable, the value printed out for x should be 1:
  
  ```
  var(x) seq assign(x,1) seq block( var(x) seq assign(x,2) ) seq put(x)
  ```
Formal Semantics:

\[
\begin{align*}
\text{(var(X),State) -->> OState :-} & \quad \% \text{ decl,} \\
& \quad \text{declarevar(X,State,OState),!}.
\end{align*}
\]

\[
\begin{align*}
\text{(assign(X,A),State) -->> OState :-} & \quad \% \text{ assignment} \\
& \quad \text{lookup(X,State,_),} \\
& \quad \text{(A,State) -->> ValA,} \\
& \quad \text{bindval(X,ValA,State,OState),!}.
\end{align*}
\]

\[
\begin{align*}
\text{(block(C),State) -->> OState :-} & \quad \% \text{block} \% \quad \text{block statement} \\
& \quad \text{pushenv(State,LocalState),} \\
& \quad \text{(C,LocalState) -->> S,} \\
& \quad \text{popenv(S,OState),!}.
\end{align*}
\]

**Note:** Each block now pushes its own binding environment on an environment stack.

**Note:** The new semantic procedures 'declarevar' and 'bindval' are necessary because declaring and binding is done differently with nested scopes.
Semantic procedures 'pushenv' and 'popenv':

% the predicate 'pushenv(+State,-FinalState)' pushes
% a new binding term list on the stack
:- dynamic pushenv/2.

pushenv(S,env([],S)) :- !.

% the predicate 'popenv(+State,-FinalState)' pops
% a binding term list off the stack
:- dynamic popenv/2.

popenv(env(_,S),S) :- !.
Looking up variable bindings in a stack of binding environments:

% the predicate 'lookup(+Variable,+State,-Value)' looks up
% the variable in the state and returns its bound value.
:- dynamic lookup/3. % modifiable predicate

lookup(_,s0,_):= fail.

lookup(X,env([],S),Val):= lookup(X,S,Val),!.

lookup(X,env([bind(Val,X)|_],_),Val).

lookup(X,env([_|Rest],S),Val):= lookup(X,env(Rest,S),Val),!.
Semantic procedure 'declarevar':

% the predicate 'declarevar(+Variable,+State,-FinalState)' declares
% a variable by inserting a new binding term into the current
% environment.
:- dynamic declarevar/3. % modifiable predicate

declarevar(X,S,env([bind(0,X)],S)) :-
    atom(S),!.

declarevar(X,env(L,S),env([bind(0,X)|L],S)) :- !.
Semantic procedure 'bindval':

% the predicate 'bindval(+Variable,+Value,+State,-FinalState)' updates
% a binding term in the state. this update is done "in place"
% in order to support global variables. the predicate has to
% search both the binding list and the stack of binding
% lists.
:- dynamic bindval/4. % modifiable predicate

bindval(_,_,s0,_ :-
    fail.

bindval(X,Val,env([],S),env([],NewS)) :-
    bindval(X,Val,S,NewS),!.

bindval(X,Val,env([bind(_,X)|L],S),env([bind(Val,X)|L],S),NewS),!.

bindval(X,Val,env([bind(V,Y)|L],S),env([bind(V,Y)|NewL],NewS)) :-
    bindval(X,Val,env(L,S),env(NewL,NewS)),!.
?- ['sem-block.pl'].
% xis.pl compiled 0.00 sec, 7,792 bytes
% preamble.pl compiled 0.00 sec, 8,956 bytes
% xis.pl compiled 0.00 sec, 148 bytes
% sem-block.pl compiled 0.00 sec, 18,192 bytes
true.

?- ((var(x) seq block( var(y) seq assign(y,1) ) seq assign(y,x)),s) --> V.
false.

?- ((var(x) seq assign(x,1) seq block( assign(x,2) ) seq put(x)),s) --> V.
x is 2
V = env([bind(2, x)], s).

?- (( var(x) seq assign(x,1) seq block( var(x) seq assign(x,2) ) seq put(x)),s) --> V.
x is 1
V = env([bind(1, x)], s).
?-